



Ahoo Engineering Group

Software Engineering

by Nguyen Xuan Huy

Institute of Information Technology,

National Center for Natural Science and Technology

Email: nxhuy@ioit.ncst.ac.vn

Duration: 45 hours



Objectives

The objectives of this chapter are to define what is meant by software engineering, to discuss which attributes are characteristic of software products and which measures need to be undertaken to assure quality. The notion of software process models is introduced and discussed.

Contents

1.1 Scope

1.1.1 The emergence of software engineering

1.1.2 The term software engineering

1.2 Quality attributes of software products

1.2.1 Software quality attributes

1.2.2 The importance of quality criteria

1.2.3 The effects of quality criteria on each other

1.2.4 Quality assurance measures

1.3 The phases of a software project

1.3.1 The classical sequential software life-cycle model

1.3.2 The waterfall model

1.3.3 The prototyping-oriented life-cycle model

1.3.4 The spiral model

1.3.5 The object-oriented life-cycle model

1.3.6 The object and prototyping-oriented life-cycle model

References and selected reading

1.1 Scope

1.1.1 The emergence of software engineering

Many persons work on producing software systems for many users.

The task description and the requirements frequently change even during the program design phase, and continue to change even after the software system has long since been in use.



The major problems encountered in development of large software systems were:

- Correctness
- Efficiency
- Mastery of complexity
- Interface specification
- Reliability
- Flexibility
- Documentation
- Maintainability
- Project organization.

Inadequate theoretical foundation and too few methodological aids were known in both the technical and the organizational realm.

Programmers' qualifications did not suffice to adequately solve the problems.

Software crisis (1965):

The growing economic importance of software production and the enormous expansion of the data processing branch lead to the demand for improved *programming techniques* becoming an increasing focus of research interests in the field of computer science.

Software engineering conferences sponsored by NATO:

- Garmisch, Germany, in 1968 [Naur 1969]
- Rome in 1969 [Buxton 1969].

Contents of the Conferences:

- Programs were defined as *industrial products*.
- The challenge was issued: to move away from the *art of programming* and toward an *engineering approach* to software development.
- Application of scientific approaches throughout software production as an integrated process.
- Research highlights:
 - Specification,
 - Methodical program development,
 - Structuring of software systems,
 - Reusability of software components,
 - Project organization,
 - Quality assurance,
 - Documentation,
 - Requirements for software development tools,
 - Automatic generation of software.



1.1.2 The term software engineering

It will be clear that the term *software engineering* involves many different issues - all are crucial to the success of large-scale software development. The topics covered in this course are intended to address all these different issues.

Boehm [Boehm 1979]:

Software Engineering: The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.

Dennis [Dennis 1975]:

Software engineering is the application of principles, skills, and art to the design and construction of programs and systems of programs.

Parnas [Parnas 1974]:

Software engineering is programming under at least one of the following two conditions:

- (1) More than one person is involved in the construction and/or use of the programs*
- (2) More than one version of the program will be produced*

Fairley [Fairley 1985]:

Software engineering is the technological and managerial discipline concerned with the systematic production and maintenance of software products that are developed and modified on time and within cost estimates.

Sommerville [Sommerville 1989]:

Software Engineering is concerned with building software systems which are large than would normally be tackled by a single individual, uses engineering principles in the development of these systems and is made up of both technical and non-technical aspects.

Pomberger and Blaschek [Pomberger 1996]:

Software engineering is the practical application of scientific knowledge for the economical production and use of high-quality software.

1.2 Quality attributes of software products

1.2.1 Software quality attributes

Software quality is a broad and important field of software engineering.

Addressed by standardization bodies:

- ISO
- ANSI
- IEEE
- ...

Software quality attributes (see Figure 1.1)

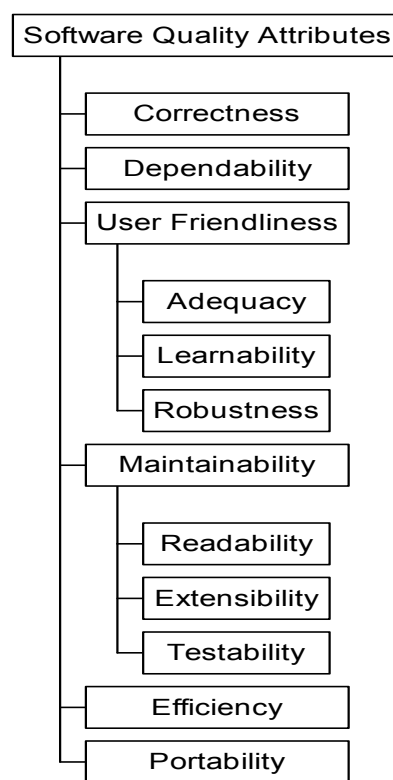


Figure 1.1 Software quality attributes

Correctness

The software system's fulfillment of the specifications underlying its development. The correctness of a software system refers to:

- Agreement of *program code* with *specifications*
- Independence of the actual application of the software system.

The correctness of a program becomes especially critical when it is embedded in a complex software system.



Example 1.1 Let p be the probability that an individual program is correct; then the probability P of correctness of the software system consisting of n programs is computed as $P = p^n$.

If n is very large, then p must be nearly 1 in order to allow P to deviate significantly from 0 [Dijkstra 1972b].

Reliability

Reliability of a software system derives from

- Correctness, and
- Availability.

The behavior over time for the fulfillment of a given specification depends on the reliability of the software system.

Reliability of a software system is defined as the probability that this system fulfills a function (determined by the specifications) for a specified number of input trials under specified input conditions in a specified time interval (assuming that hardware and input are free of errors).

A software system can be seen as reliable if this test produces a low *error rate* (i.e., the probability that an error will occur in a specified time interval.)

The error rate depends on the frequency of inputs and on the probability that an individual input will lead to an error.

User friendliness:

- Adequacy
- Learnability
- Robustness

Adequacy

Factors for the requirement of **Adequacy**:

1. The input required of the user should be limited to only what is necessary. The software system should expect information only if it is necessary for the functions that the user wishes to carry out. The software system should enable flexible data input on the part of the user and should carry out plausibility checks on the input. In dialog-driven software systems, we vest particular importance in the uniformity, clarity and simplicity of the dialogs.
2. The performance offered by the software system should be adapted to the wishes of the user with the consideration given to extensibility; i.e., the functions should be limited to these in the specification.
3. The results produced by the software system:
The results that a software system delivers should be output in a clear and well-structured form and be easy to interpret. The software system should afford the user flexibility with respect to the scope, the degree of detail, and the form of



presentation of the results. Error messages must be provided in a form that is comprehensible for the user.

Learnability

Learnability of a software system depends on:

- The design of user interfaces
- The clarity and the simplicity of the user instructions (tutorial or user manual).

The user interface should present information as close to reality as possible and permit efficient utilization of the software's failures.

The user manual should be structured clearly and simply and be free of all dead weight. It should explain to the user what the software system should do, how the individual functions are activated, what relationships exist between functions, and which exceptions might arise and how they can be corrected. In addition, the user manual should serve as a reference that supports the user in quickly and comfortably finding the correct answers to questions.

Robustness

Robustness reduces the impact of operational mistakes, erroneous input data, and hardware errors.

A software system is robust if the consequences of an error in its operation, in the input, or in the hardware, in relation to a given application, are inversely proportional to the probability of the occurrence of this error in the given application.

- Frequent errors (e.g. erroneous commands, typing errors) must be handled with particular care
- Less frequent errors (e.g. power failure) can be handled more laxly, but still must not lead to irreversible consequences.

Maintainability

Maintainability = suitability for debugging (localization and correction of errors) and for modification and extension of functionality.

The maintainability of a software system depends on its:

- Readability
- Extensibility
- Testability

Readability

Readability of a software system depends on its:

- Form of representation
- Programming style
- Consistency
- Readability of the implementation programming languages



- Structuredness of the system
- Quality of the documentation
- Tools available for inspection

Extensibility

Extensibility allows required modifications at the appropriate locations to be made without undesirable side effects.

Extensibility of a software system depends on its:

- Structuredness (modularity) of the software system
- Possibilities that the implementation language provides for this purpose
- Readability (to find the appropriate location) of the code
- Availability of comprehensible program documentation

Testability

Testability: suitability for allowing the programmer to follow program execution (run-time behavior under given conditions) and for debugging.

The testability of a software system depends on its:

- Modularity
- Structuredness

Modular, well-structured programs prove more suitable for systematic, stepwise testing than monolithic, unstructured programs.

Testing tools and the possibility of formulating consistency conditions (assertions) in the source code reduce the testing effort and provide important prerequisites for the extensive, systematic testing of all system components.

Efficiency

Efficiency: ability of a software system to fulfill its purpose with the best possible utilization of all necessary resources (time, storage, transmission channels, and peripherals).

Portability

Portability: the ease with which a software system can be adapted to run on computers other than the one for which it was designed.

The portability of a software system depends on:

- Degree of hardware independence
- Implementation language
- Extent of exploitation of specialized system functions
- Hardware properties
- Structuredness: System-dependent elements are collected in easily interchangeable program components.

A software system can be said to be portable if the effort required for porting it proves significantly less than the effort necessary for a new implementation.



1.2.2 The importance of quality criteria

The quality requirements encompass all levels of software production.

Poor quality in intermediate products always proves detrimental to the quality of the final product.

- Quality attributes that affect the end product
- Quality attributes that affect intermediate products

Quality of *end products* [Bons 1982]:

- Quality attributes that affect their *application*: These influence the suitability of the product for its intended application (correctness, reliability and user friendliness).
- Quality attributes related to their *maintenance*: These affect the suitability of the product for functional modification and extensibility (readability, extensibility and testability).
- Quality attributes that influence their *portability*: These affect the suitability of the product for porting to another environment (portability and testability).

Quality attributes of *intermediate products*:

- Quality attributes that affect the transformation: These affect the suitability of an intermediate product for immediate transformation to a subsequent (high-quality) product (correctness, readability and testability).
- Quality attributes that affect the quality of the end product: These directly influence the quality of the end product (correctness, reliability, adequacy, readability, extensibility, testability, efficiency and portability).



1.2.3 The effects of quality criteria on each other

Attributes \ Effect on										
	Correctness	Dependability	Adequacy	Learnability	Robustness	Readability	Modifiability/extensibility	Testability	Efficiency	Portability
Correctness		+	0	0	+	0	0	0	0	0
Dependability	0		0	0	+	0	0	0	-	0
Adequacy	0	0		+	0	0	0	0	+	-
Learnability	0	0	0		0	0	0	0	-	0
Robustness	0	+	+	0		0	0	+	-	0
Readability	+	+	0	0	+		+	+	-	+
Modifiability/extensibility	+	+	0	0	+	0		+	-	+
Testability	+	+	0	0	+	0	+		-	+
Efficiency	-	-	+	-	-	-	-	-		-
Portability	0	0	-	0	0	0	+	0	-	

Table 1.1 Mutual effects between quality criteria (“+”: positive effect, “-“: negative effect, “0”: no effect)

1.2.4 Quality assurance measures

The most important measures are:

1. Constructive measures:
 - Consistent application of methods in all phases of the development process
 - Use of adequate development tools
 - Software development on the basis of high-quality semifinished products
 - Consistent maintenance of development documentation
2. Analytical measures:
 - Static program analysis
 - Dynamic program analysis

- Systematic choice of adequate test cases
 - Consistent logging of analysis results
3. Organizational measures:
- Continuous education of product developers
 - Institutionalization of quality assurance

1.3 The phases of a software project

Software projects are divided into individual phases. These phases collectively and their chronological sequence are termed the *software life cycle*.

Software life cycle: a time span in which a software product is developed and used, extending to its retirement.

The cyclical nature of the model expresses the fact that the phases can be carried out repeatedly in the development of a software product.

1.3.1 The classical sequential software life-cycle model

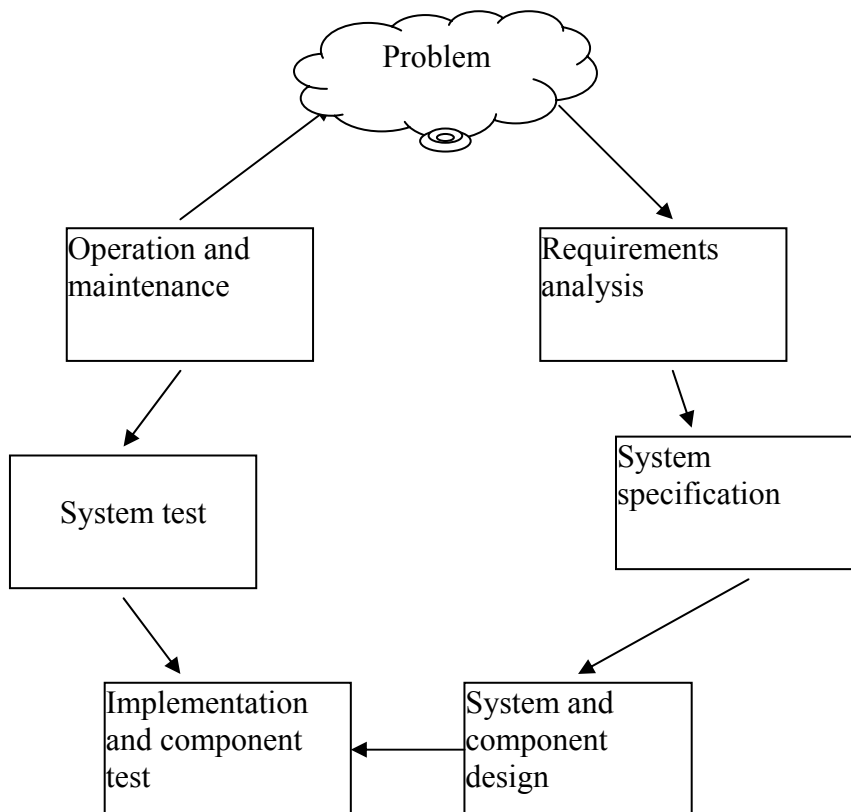


Figure 1.2 The classical sequential software life-cycle model

Requirements analysis and planning phase

Goal:



- Determining and documenting:
 - ❖ Which steps need to be carried out,
 - ❖ The nature of their mutual effects,
 - ❖ Which parts are to be automated, and
 - ❖ Which recourses are available for the realization of the project.

Important activities:

- Completing the requirements analysis,
- Delimiting the problem domain,
- Roughly sketching the components of the target system,
- Making an initial estimate of the scope and the economic feasibility of the planned project, and
- Creating a rough project schedule.

Products:

- User requirements,
- Project contract, and
- Rough project schedule.

System specification phase

Goal:

- a contract between the client and the software producer (precisely specifies what the target software system must do and the premises for its realization.)

Important activities:

- Specifying the system,
- Compiling the requirements definition,
- Establishing an exact project schedule,
- Validating the system specification, and
- Justifying the economic feasibility of the project.

Products:

- Requirements definition, and
- Exact project schedule.

System and components design

Goal:

- Determining which system components will cover which requirements in the system specification, and
- How these system components will work together.

Important activities:



- Designing system architecture,
- Designing the underlying logical data model,
- Designing the algorithmic structure of the system components, and
- Validating the system architecture and the algorithms to realize the individual system components.

Products:

- Description of the logical data model,
- Description of the system architecture,
- Description of the algorithmic structure of the system components, and
- Documentation of the design decisions.

Implementation and component test

Goal:

- Transforming the products of the design phase into a form that is executable on a computer.

Important activities:

- Refining the algorithms for the individual components,
- Transferring the algorithms into a programming language (coding),
- Translating the logical data model into a physical one,
- Compiling and checking the syntactical correctness of the algorithm, and
- Testing, and syntactically and semantically correcting erroneous system components.

Products:

- Program code of the system components,
- Logs of the component tests, and
- Physical data model.

System test

Goal:

- Testing the mutual effects of system components under conditions close to reality,
- Detecting as many errors as possible in the software system, and
- Assuring that the system implementation fulfills the system specification.

Operation and maintenance

Task of software maintenance:



- Correcting errors that are detected during actual operation, and
- Carrying out system modifications and extensions.

This is normally the longest phase of the software life cycle.

Two important additional aspects:

- Documentation, and
- Quality assurance.

During the development phases the *documentation* should enable communication among the persons involved in the development; upon completion of the development phases it supports the utilization and maintenance of the software product.

Quality assurance encompasses analytical, design and organizational measures for quality planning and for fulfilling quality criteria such as correctness, reliability, user friendliness, maintainability, efficiency and portability.

1.3.2 The waterfall model

Developed in the 1970s [Royce 1970].

The Waterfall Model represent an experience-based refinement of the classical sequential software life-cycle model.

Two extensions:

1. It introduces iteration between the phases along with the restriction of providing iterations, if possible, only between successive phases in order to reduce the expense of revision that results from iterations over multiple phases.
2. It provides for validation of the phase outputs in the software life cycle.

This approach modifies the strictly sequential approach prescribed by the classical life-cycle model and advances an incremental development strategy. Incorporating a stepwise development strategy for the system specifications and the system architecture as well as phase-wise validation helps to better manage the effects of poor decisions and to make the software development process more controllable.

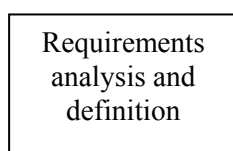




Figure 1.3 The Waterfall model

1.3.3 The prototyping-oriented life-cycle model

Developed in the 1990s [Pomberger 1991].

A prototyping-oriented software development strategy does not differ fundamentally from the classical phase-oriented development strategy. These strategies are more complementary than alternative.

New aspects:

- The phase model is seen not as linear but as iterative, and
- It specifies where and how these iterations are not only possible but necessary.

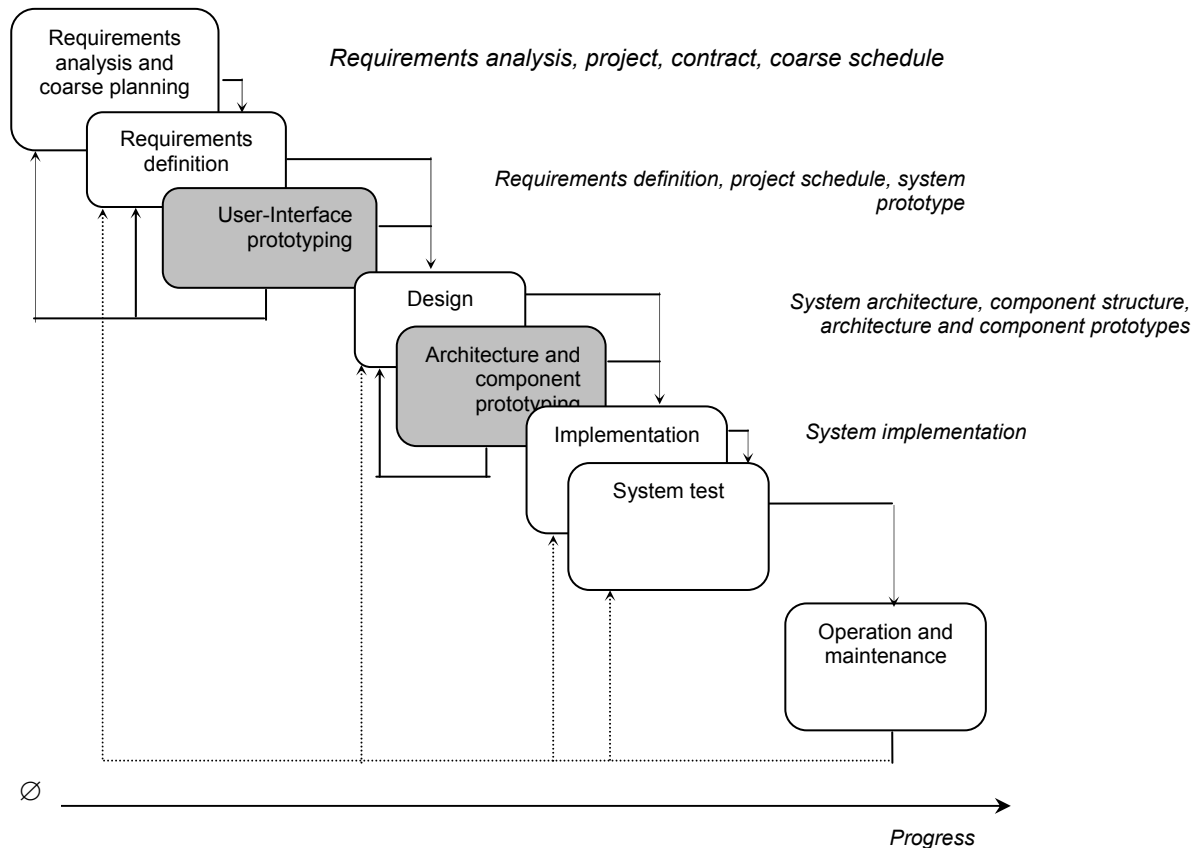


Figure 1.4 Prototyping-oriented software life cycle

Prototype construction is an iterative process (Figure 1.4). First a prototype is produced on the basis of the results of preceding activities. During the specification phase a prototype of the user interface is developed that encompasses significant parts of the functional requirements on the target software system. Experiments are conducted on this prototype that reflect real conditions of actual use in order to determine whether the client's requirements are fulfilled. Under conditions close to reality, software developers and users can test whether the system model contains errors, whether it meets the client's preconceptions, and whether modifications are necessary. This reduces the risk of erroneous or incomplete system specifications and creates a significantly better starting point for subsequent activities.

1.3.4 The spiral model

Developed in 1988 [Boehm 1988].

The spiral model is a software development model that combines the above models or includes them as special cases. The model makes it possible to choose the most suitable approach for a given project. Each cycle encompasses the same sequence of steps for each part of the target product and for each stage of completion.

A spiral cycle begins with the establishment of the following point:

- Goals for and requirements on the (sub)product



- Alternatives to realization of the (sub)product
- Constraints and restrictions

The next step evaluates the proposed solution variant with respect to the project goals and applicable constraints, emphasizing the detection of risks and uncertainties. If such are found, measures and strategies are considered to reduce these risks and their effects.

Important aspects of the spiral model:

- Each cycle has its own validation step that includes all persons participating in the project and the affected future users or organizational unit,
- Validation encompasses all products emanating from the cycle, including the planning for the next cycle and the required resources.

1.3.5 The object-oriented life-cycle model (Figure 1.5)

The usual division of a software project into phases remains intact with the use of object-oriented techniques.

The requirements analysis stage strives to achieve an understanding of the client's application domain.

The tasks that a software solution must address emerge in the course of requirements analysis.

The requirements analysis phase remains completely independent of an implementation technique that might be applied later.

In the system specification phase the requirements definition describes *what* the software product must do, but not *how* this goal is to be achieved.

One point of divergence from conventional phase models arises because implementation with object-oriented programming is marked by the assembly of already existing components.

The advantages of object-oriented life-cycle model:

- Design no longer be carried out independently of the later implementation because during the design phase we must consider which components are available for the solution of the problem. Design and implementation become more closely associated, and even the choice of a different programming language can lead to completely different program structures.
- The duration of the implementation phase is reduced. In particular, (sub)products become available much earlier to allow testing of the correctness of the design. Incorrect decisions can be recognized and corrected earlier. This makes for closer feedback coupling of the design and implementation phases.
- The class library containing the reusable components must be continuously maintained. Saving at the implementation end is partially lost as they are reinvested in this maintenance. A new job title emerges, the class librarian, who is responsible for ensuring the efficient usability of the class library.
- During the test phase, the function of not only the new product but also of the reused components is tested. Any deficiencies in the latter must be



Ahoo Engineering Group

documented exactly. The resulting modifications must be handled centrally in the class library to ensure that they impact on other projects, both current and future.

- Newly created classes must be tested for their general usability. If there is a chance that a component could be used in other projects as well, it must be included in the class library and documented accordingly. This also means that the new class must be announced and made accessible to other programmers who might profit from it. This places new requirements on the in-house communication structures.

The class library serves as a tool that extends beyond the scope of an individual project because classes provided by one project can increase productivity in subsequent projects.

The actual software life cycle recurs when new requirements arise in the company that initiate a new requirements analysis stage.

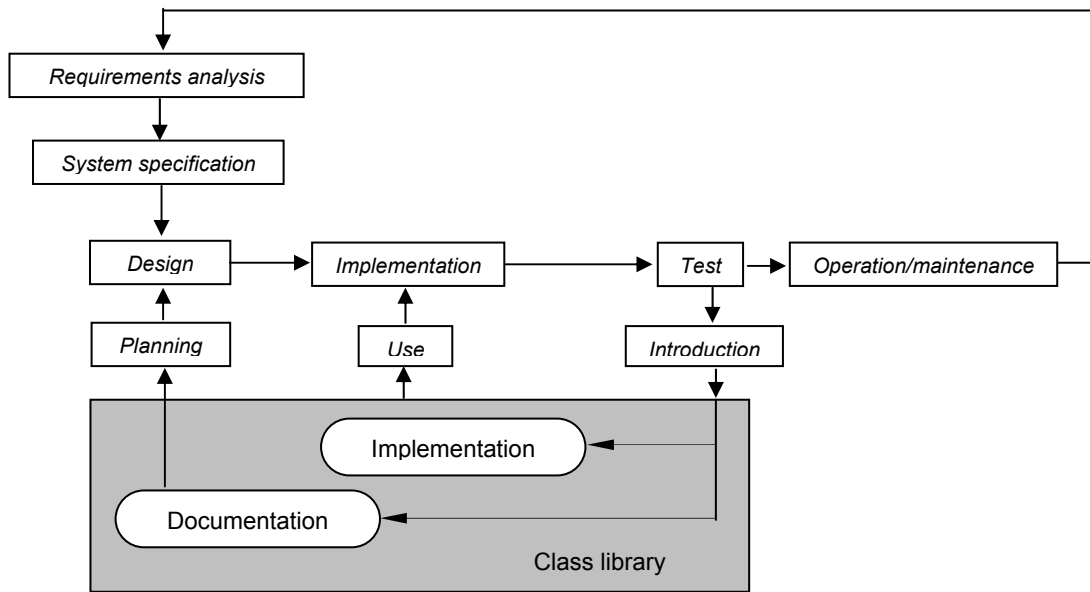


Figure 1.5 Object-oriented life-cycle model

1.3.6 The object and prototyping-oriented life-cycle model (Figure 1.6)

The specification phase steadily creates new prototypes. Each time we are confronted with the problem of having to modify or enhance existing prototypes. If the prototypes were already implemented with object-oriented technology, then modifications and extensions are particularly easy to carry out. This allows an abbreviation of the specification phase, which is particularly important when proposed solutions are repeatedly discussed with the client. With such an approach it is not important whether the prototype serves solely for specification purposes or whether it is to be incrementally developed to the final product. If no prototyping tools are available, object-oriented programming can serve as a substitute tool for modeling user interfaces. This particularly applies if an extensive class library is available for user interface elements.

For incremental prototyping (i.e. if the product prototype is to be used as the basis for the implementation of the product), object-oriented programming also proves to be a suitable medium. Desired functionality can be added stepwise to the prototypes without having to change the prototype itself. This results in a clear distinction between the user interface modeled in the specification phase and the actual functionality of the program. This is particularly important for the following reasons:

- This assures that the user interface is not changed during the implementation of the program functionality. The user interface developed in collaboration with the client remains as it was defined in the specification phase.
- In the implementation of the functionality, each time a subtask is completed, a more functional prototype results, which can be tested (preferably together with the client) and compared with the specifications. During test runs situations sometimes arise that require rethinking the user interface. In such cases the software life cycle retreats one step and a new user interface prototype is constructed.

Since the user interface and the functional parts of the program are largely decoupled, two cycles result that share a common core. The integration of the functional classes and the user interface classes creates a prototype that can be tested and validated. This places new requirements on the user interface and/or the functionality, so that the cycle begins.

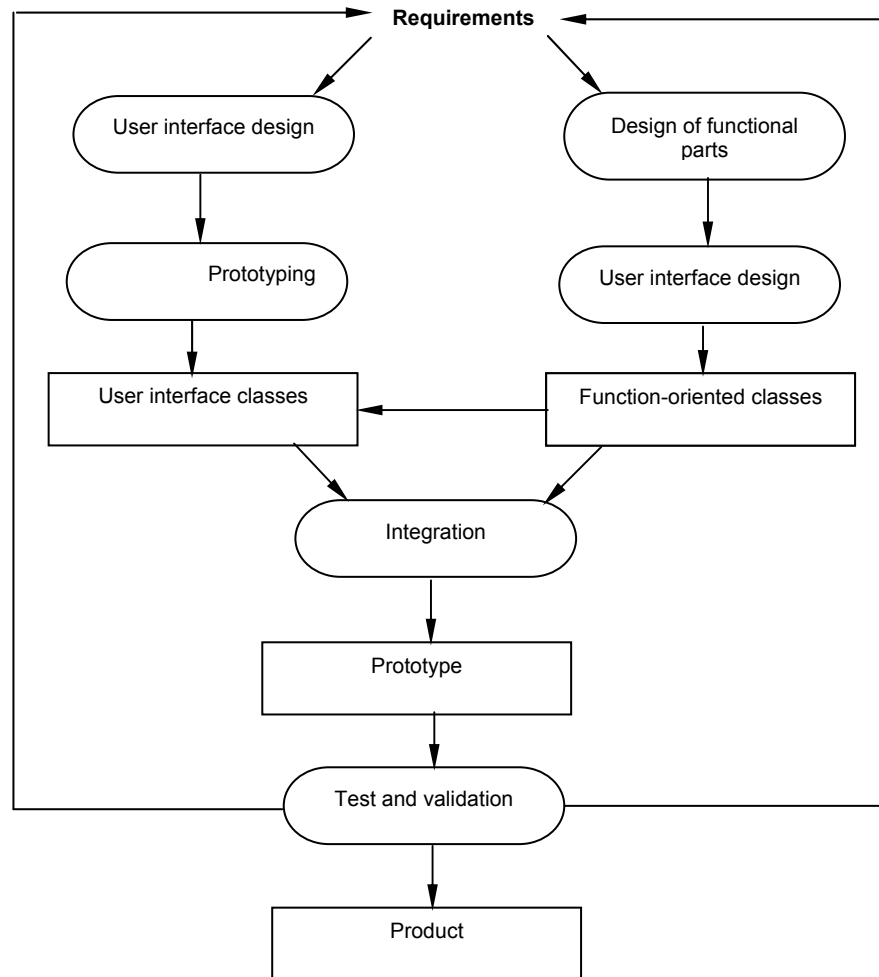


Figure 1.6 Software development with prototyping and object-orientation

References and selected reading

- [Boehm 1979] Boehm B. W., Software Engineering, in: *Classics in Software Engineering*, Yourdon Press, 1979.
- [Buxton 1969] Buxton J. N., Randell B. (Eds.), Software Engineering Techniques, *Report on a Conference*, Rome, Brussels: NATO Scientific Affairs Division, 1969



- [Boehm 1988] Boehm B. V., A Spiral Model of Software Development and Enhancement, *IEEE Computer* 21, 1988
- [Bons 1982] Bons H., van Megen R., Zur Festlegung von Qualitätszielen als Grundlage der Qualitätsplanung und-Kontrolle, *German Chapter of the ACM, Bericht* 9, 1982
- [Dennis 1975] Dennis J. B., *The Design and Construction of Software Systems*, in: *Software Engineering As An Advanced Course*, Springer, 1975
- [Dijkstra 1972b] Dijkstra E. W., *Notes on Structured Programming*, Academic Press, 1972
- [Fairley 1985] Fairley R., *Software Engineering Concepts*, McGraw-Hill, 1985
- [Naur 1969] Naur P., Randell B. (Eds.), *Software Engineering, Report on a Conference*, Garmisch, 1968, Brussels, NATO Scientific Affairs Division, 1969
- [Parnas 1974] Parnas D. L., *Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs*, Lecture Notes in Computer Science, Programming Methodology, Springer, 1974
- [Pomberger 1991] Pomberger G. et al., Prototyping-Oriented Software Development – Concepts and Tools, *Structured Programming*, Vol. 12, Nr. 1, 1991
- [Pomberger 1996] Pomberger G. and Blaschek G., *Object Orientation and Prototyping in Software Engineering*, Translated by Bach R., Prentice Hall, 1996
- [Royce 1970] Royce W., Managing the Development of Large Software Systems: Concepts, *WESCON Proceedings*, 1970
- [Pressman 1997] Pressman R., *Software Engineering: A practitioner's Approach*, (4th ed.), McGraw-Hill, 1997
- [Sommerville 1996] Sommerville I., *Software Engineering* (5th ed.), Addison-Wesley, 1996.